Computer Architecture

Chapter 3 - Instruction-Level Parallelism

Software Techniques - Code Scheduling

The straightforward RISC-V code, not scheduled for the pipeline, looks like this:

```
Loop:  fld     f0,0(x1)      //f0=array element
       fadd.d  f4,f0,f2      //add scalar in f2
       fsd     f4,0(x1)      //store result
       addi    x1,x1,−8      //decrement pointer
                             //8 bytes (per DW)
       bne     x1,x2,Loop    //branch x1≠x2
```

Without any scheduling, the loop will execute as follows, taking nine cycles:

```
                                    Clock cycle issued
Loop:  fld     f0,0(x1)                    1
       stall                               2
       fadd.d  f4,f0,f2                    3
       stall                               4
       stall                               5
       fsd     f4,0(x1)                    6
       addi    x1,x1,−8                    7
       bne     x1,x2,Loop                  8
```

We can schedule the loop to obtain only two stalls and reduce the time to seven cycles:

```
Loop:  fld     f0,0(x1)
       addi    x1,x1,−8
       fadd.d  f4,f0,f2
       stall
       stall
       fsd     f4,8(x1)
       bne     x1,x2,Loop
```

The stalls after `fadd.d` are for use by the `fsd`, and repositioning the `addi` prevents the stall after the `fld`.

Loop Unrolling

- Unrolled 4 times

Here is the result after merging the `addi` instructions and dropping the unnecessary `bne` operations that are duplicated during unrolling. Note that `x2` must now be set so that `Regs[x2]+32` is the starting address of the last four elements.

```
Loop:  fld     f0,0(x1)
       fadd.d  f4,f0,f2
       fsd     f4,0(x1)      //drop addi & bne
       fld     f6,-8(x1)
       fadd.d  f8,f6,f2
       fsd     f8,-8(x1)     //drop addi & bne
       fld     f0,-16(x1)
       fadd.d  f12,f0,f2
       fsd     f12,-16(x1)   //drop addi & bne
       fld     f14,-24(x1)
       fadd.d  f16,f14,f2
       fsd     f16,-24(x1)
       addi    x1,x1,-32
       bne     x1,x2,Loop
```

Unrolled and Scheduled

Example Show the unrolled loop in the previous example after it has been scheduled for the pipeline with the latencies in Figure 3.2.

```
Loop:  fld     f0,0(x1)
       fld     f6,-8(x1)
       fld     f0,-16(x1)
       fld     f14,-24(x1)
       fadd.d  f4,f0,f2
       fadd.d  f8,f6,f2
       fadd.d  f12,f0,f2
       fadd.d  f16,f14,f2
       fsd     f4,0(x1)
       fsd     f8,-8(x1)
       fsd     f12,16(x1)
       fsd     f16,8(x1)
       addi    x1,x1,-32
       bne     x1,x2,Loop
```

The execution time of the unrolled loop has dropped to a total of 14 clock cycles, or 3.5 clock cycles per element, compared with 8 cycles per element before any unrolling or scheduling and 6.5 cycles when unrolled but not scheduled.

Dynamic Scheduling - Tomasulo's Algorithm

Some issues with pipelining

- Memory - unpredictable retrieval speed - cache behavior
- A stall in one instruction causes entire pipeline to stall
- Long-executing instructions cause long stalls
- Independent instructions get stalled, even though they could execute
- Out-of-order execution might cause WAR and WAW hazards (name dependences and anti-dependences)
- Doesn't allow deep speculation
- Not easy to add additional functional units (adders, multipliers, etc.)

Robert Tomasulo - invented technique for IBM 360/91 FP Unit

The “Big Mac” Analogy

- The McDonalds drive-up window is a pipeline (“In-order-issue”)
- If a customer’s Big Mac (“operand”) is not ready, it stalls the pipeline
- If the wait is long, customer is asked to pull out of the way and wait in a separate parking space (“reservation station”). This allows the pipeline to move again (“eliminates the stall”)
- When Big Mac is done, employee delivers it to customer
- “Out-of-order” completion.

New plan for instruction execution

- *Issue* (or *dispatch*) - instructions are submitted in-order for execution. Instruction waits in a *reservation station* until operands are available.
- *Execute* - when all operands are available, instruction is submitted to an appropriate *functional unit.*
- *Write Result* - when result is available, write it to the *Common Data Bus*, which distributes result to reservation station operands and register

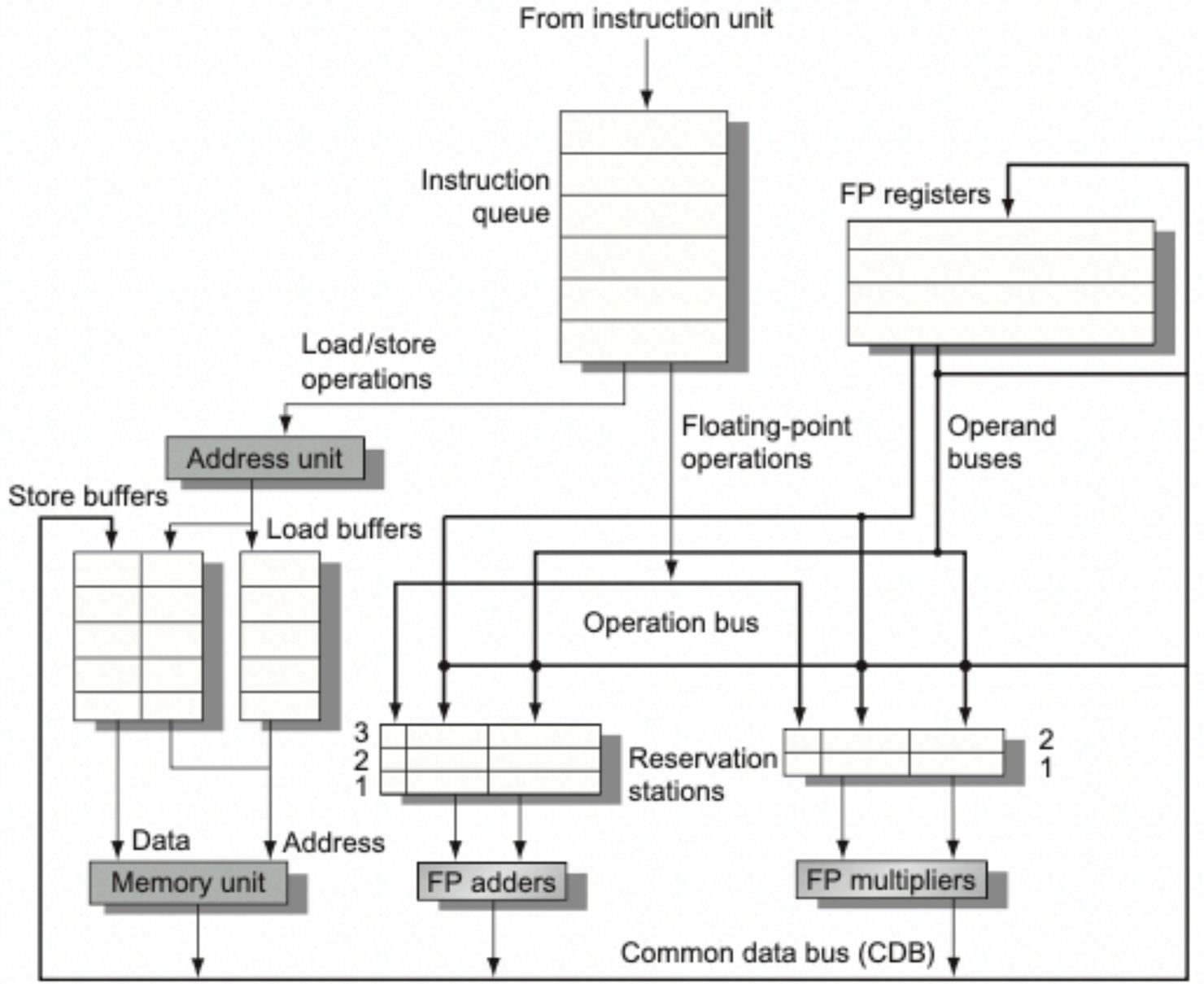

Figure 3.10 The basic structure of a RISC-V floating-point unit using Tomasulo's algorithm. Instruction

Reservation Station Fields

- *Op* - operation to perform on operands *S1* and *S2*
- *Qj, Qk* - Reservation stations that will produce operand values for *S1* and *S2*
- *Vj, Vk* - Value of source operands, if known already. If known, corresponding *Q* is blank
- *A* - holds memory address (*EA* - Effective Address) if operand comes from memory
- *Busy* - Indicates that reservation station is busy

Register File

- *Qi* - Reservation Station that will produce the result for the register, blank if result is known

Load/Store Buffers

- A - holds *EA* of memory value

Example Code

```
1.   fld     f6,32(x2)
2.   fld     f2,44(x3)
3.   fmul.d  f0,f2,f4
4.   fsub.d  f8,f2,f6
5.   fdiv.d  f0,f0,f6
6.   fadd.d  f6,f8,f2
```

Tomasulo Example 1

Show what the Tomasulo machine looks like for the previous code sequence when only the first load has completed and written its result.

Instruction	Instruction Status		
	Issue	Execute	Write Result

Reservation Status							
Name	Busy?	Op	Vj	Vk	Qj	Qk	A
LOAD1							
LOAD2							
ADD1							
ADD2							
ADD3							
MULT1							
MULT2							

Regs								
Field	F0	F2	F4	F6	F8	F10	F12	F14
Qi								

		Instruction status		
Instruction		Issue	Execute	Write result
`fld`	`f6,32(x2)`	√	√	√
`fld`	`f2,44(x3)`	√	√	
`fmul.d`	`f0,f2,f4`	√		
`fsub.d`	`f8,f2,f6`	√		
`fdiv.d`	`f0,f0,f6`	√		
`fadd.d`	`f6,f8,f2`	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					`44 + Regs[x3]`
Add1	Yes	SUB		`Mem[32 + Regs[x2]]`	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		`Regs[f4]`	Load2		
Mult2	Yes	DIV		`Mem[32 + Regs[x2]]`	Mult1		

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	...	f30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Figure 3.11 Reservation stations and register tags shown when all of the instructions have issued but only the first load instruction has completed and written its result to the CDB. The second load has completed effective

Example Continues

Using the same code segment as before, show what the status would be when the `fmul.d` is ready to write its result.

Instruction		Instruction status		
		Issue	Execute	Write result
fld	f6,32(x2)	√	√	√
fld	f2,44(x3)	√	√	√
fmul.d	f0,f2,f4	√	√	
fsub.d	f8,f2,f6	√	√	√
fdiv.d	f0,f0,f6	√		
fadd.d	f6,f8,f2	√	√	√

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL	Mem[44 + Regs[x3]]	Regs[f4]			
Mult2	Yes	DIV		Mem[32 + Regs[x2]]	Mult1		

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	...	f30
Qi	Mult1					Mult2			

Figure 3.12 Multiply and divide are the only instructions not finished.

Example 2 - A Loop-based Example

```
Loop:  fld     f0,0(x1)
       fmul.d  f4,f0,f2
       fsd     f4,0(x1)
       addi    x1,x1,-8
       bne     x1,x2,Loop  // branches if x1≠x2
```

Instruction	Instruction Status		
	Issue	Execute	Write Result

Reservation Status							
Name	Busy?	Op	Vj	Vk	Qj	Qk	A
LOAD1							
LOAD2							
ADD1							
ADD2							
ADD3							
MULT1							
MULT2							

Regs								
Field	F0	F2	F4	F6	F8	F10	F12	F14
Qi								

Instruction		From iteration	Issue	Execute	Write result
`fld`	`f0,0(x1)`	1	√	√	
`fmul.d`	`f4,f0,f2`	1	√		
`fsd`	`f4,0(x1)`	1	√		
`fld`	`f0,0(x1)`	2	√	√	
`fmul.d`	`f4,f0,f2`	2	√		
`fsd`	`f4,0(x1)`	2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Yes	Load					`Regs[x1] + 0`
Load2	Yes	Load					`Regs[x1] – 8`
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		`Regs[f2]`	Load1		
Mult2	Yes	MUL		`Regs[f2]`	Load2		
Store1	Yes	Store	`Regs[x1]`			Mult1	
Store2	Yes	Store	`Regs[x1] – 8`			Mult2	

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	…	f30
Qi	Load2		Mult2						

Figure 3.14 Two active iterations of the loop with no instruction yet completed. Entries in the multiplier reservation stations indicate that the outstanding loads are the sources. The store reservation stations indicate that the multiply destination is the source of the value to store.

Instruction state	Wait until	Action or bookkeeping
Issue FP operation	Station r empty	`if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Q ← r;`
Load or store	Buffer r empty	`if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;`
Load only		`RegisterStat[rt].Qi ← r;`
Store only		`if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`
Execute FP operation	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute result: operands are in Vj and Vk
Load/storestep 1	RS[r].Qj = 0 & r is head of load-store queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 complete	Read from `Mem[RS[r].A]`
Write result FP operation or load	Execution complete at r & CDB available	`∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;`
Store	Execution complete at r & RS[r].Qk = 0	`Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;`

Figure 3.13 Steps in the algorithm and what is required for each step. For the issuing instruction, rd is the des-

Speculative Execution

- Allow an instruction to Write Result, but don’t commit until previous instruction commits.
- Execute both branches of an IF statement, but don’t commit until the correct branch is known.
- Add another execution step:
  - 1. Issue
  - 2. Execute
  - 3. Write Result
  - 4. *Commit (*or *graduation)*
- Instructions can execute *out-of-order*, but must commit *in-order*, so add a *Reorder Buffer* (*ROB*)

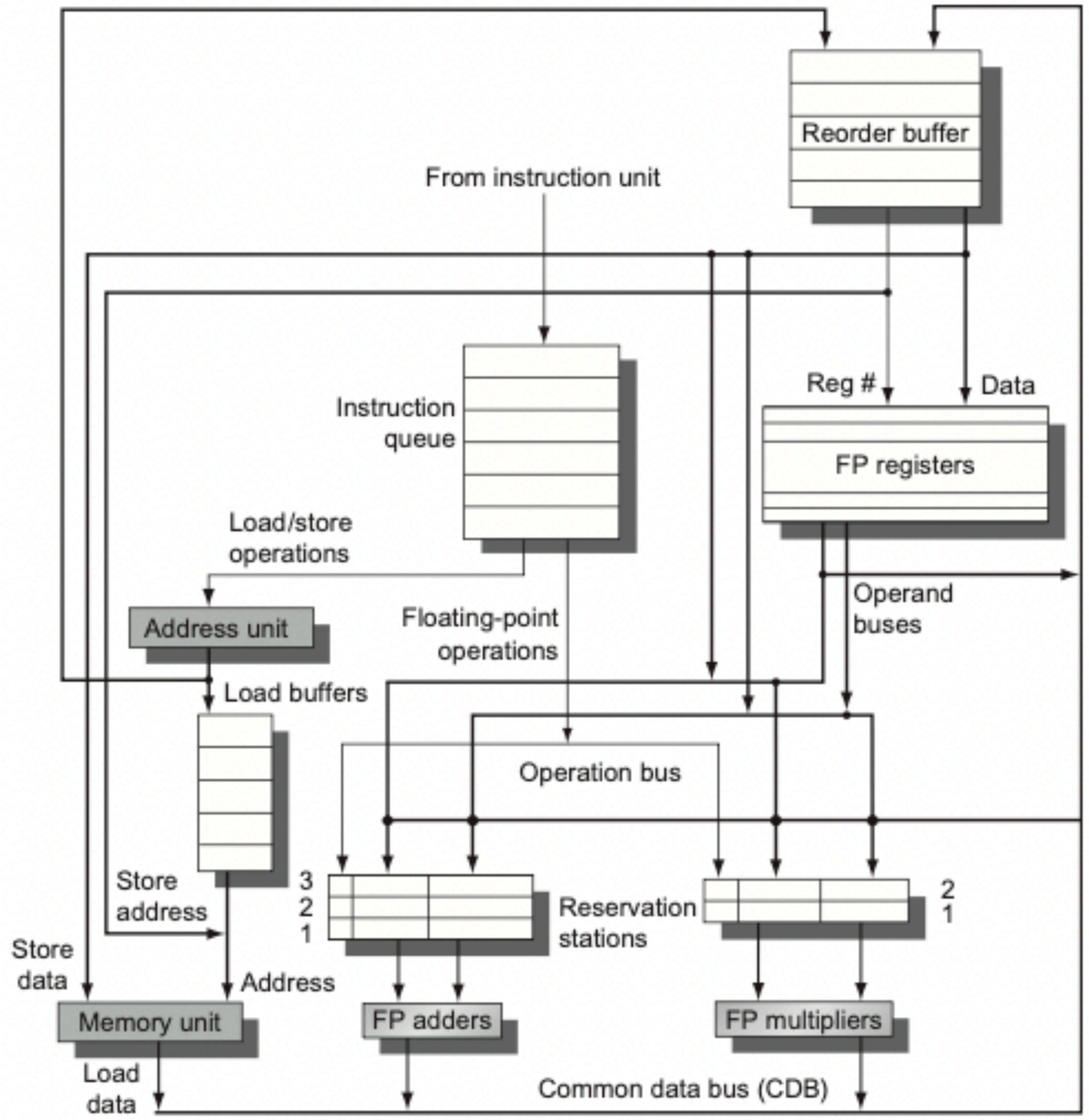

Figure 3.15 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation.

	Reorder buffer				
Entry	Busy	Instruction	State	Destination	Value
1	No	`fld f6,32(x2)`	Commit	f6	`Mem[32 + Regs[x2]]`
2	No	`fld f2,44(x3)`	Commit	f2	`Mem[44 + Regs[x3]]`
3	Yes	`fmul.d f0,f2,f4`	Write result	f0	#2 × `Regs[f4]`
4	Yes	`fsub.d f8,f2,f6`	Write result	f8	#2 – #1
5	Yes	`fdiv.d f0,f0,f6`	Execute	f0	
6	Yes	`fadd.d f6,f8,f2`	Write result	f6	#4 + #2

	Reservation stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	`fmul.d`	`Mem[44 + Regs[x3]]`	`Regs[f4]`			#3	
Mult2	Yes	`fdiv.d`		`Mem[32 + Regs[x2]]`	#3		#5	

	FP register status									
Field	f0	f1	f2	f3	f4	f5	f6	f7	f8	f10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	…	Yes	Yes

Figure 3.16 At the time the `fmul.d` is ready to commit, only the two `fld` instructions have committed, although several others have completed execution. The `fmul.d` is at the head of the ROB, and the two `fld` instructions are

Multiple Issue + Static Scheduling

More than one instruction issued per cycle - superscalar

- Statically scheduled superscalar processors
  - Two (or more) identical pipelines.
  - One regular pipeline + floating point pipeline
- Very Long Instruction Word (VLIW) processors
- Dynamically scheduled superscalar processors
- Allows a CPI < 1.0

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Figure 3.19 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of

VLIW Example

Example Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop `x[i] = x[i] + s` (see page 158 for the RISC-V code) for such a processor. Unroll as many times as necessary to eliminate any stalls.

```
Loop:  fld     f0,0(x1)      //f0=array element
       fadd.d  f4,f0,f2      //add scalar in f2
       fsd     f4,0(x1)      //store result
       addi    x1,x1,-8      //decrement pointer
                             //8 bytes (per DW)
       bne     x1,x2,Loop    //branch x1≠x2
```

178

```
Loop:  fld     f0,0(x1)
       fld     f6,-8(x1)
       fld     f0,-16(x1)
       fld     f14,-24(x1)
       fadd.d  f4,f0,f2
       fadd.d  f8,f6,f2
       fadd.d  f12,f0,f2
       fadd.d  f16,f14,f2
       fsd     f4,0(x1)
       fsd     f8,-8(x1)
       fsd     f12,16(x1)
       fsd     f16,8(x1)
       addi    x1,x1,-32
       bne     x1,x2,Loop
```

f10

-16

-24

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
fld f0,0(x1)	fld f6,-8(x1)			
fld f10,-16(x1)	fld f14,-24(x1)			
fld f18,-32(x1)	fld f22,-40(x1)	fadd.d f4,f0,f2	fadd.d f8,f6,f2	
fld f26,-48(x1)		fadd.d f12,f0,f2	fadd.d f16,f14,f2	
		fadd.d f20,f18,f2	fadd.d f24,f22,f2	
fsd f4,0(x1)	fsd f8,-8(x1)	fadd.d f28,f26,f24		
fsd f12,-16(x1)	fsd f16,-24(x1)			addi x1,x1,-56
fsd f20,24(x1)	fsd f24,16(x1)			
fsd f28,8(x1)				bne x1,x2,Loop

Figure 3.20 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9

ILP Using Dynamic Scheduling+Multiple Issue+Speculation

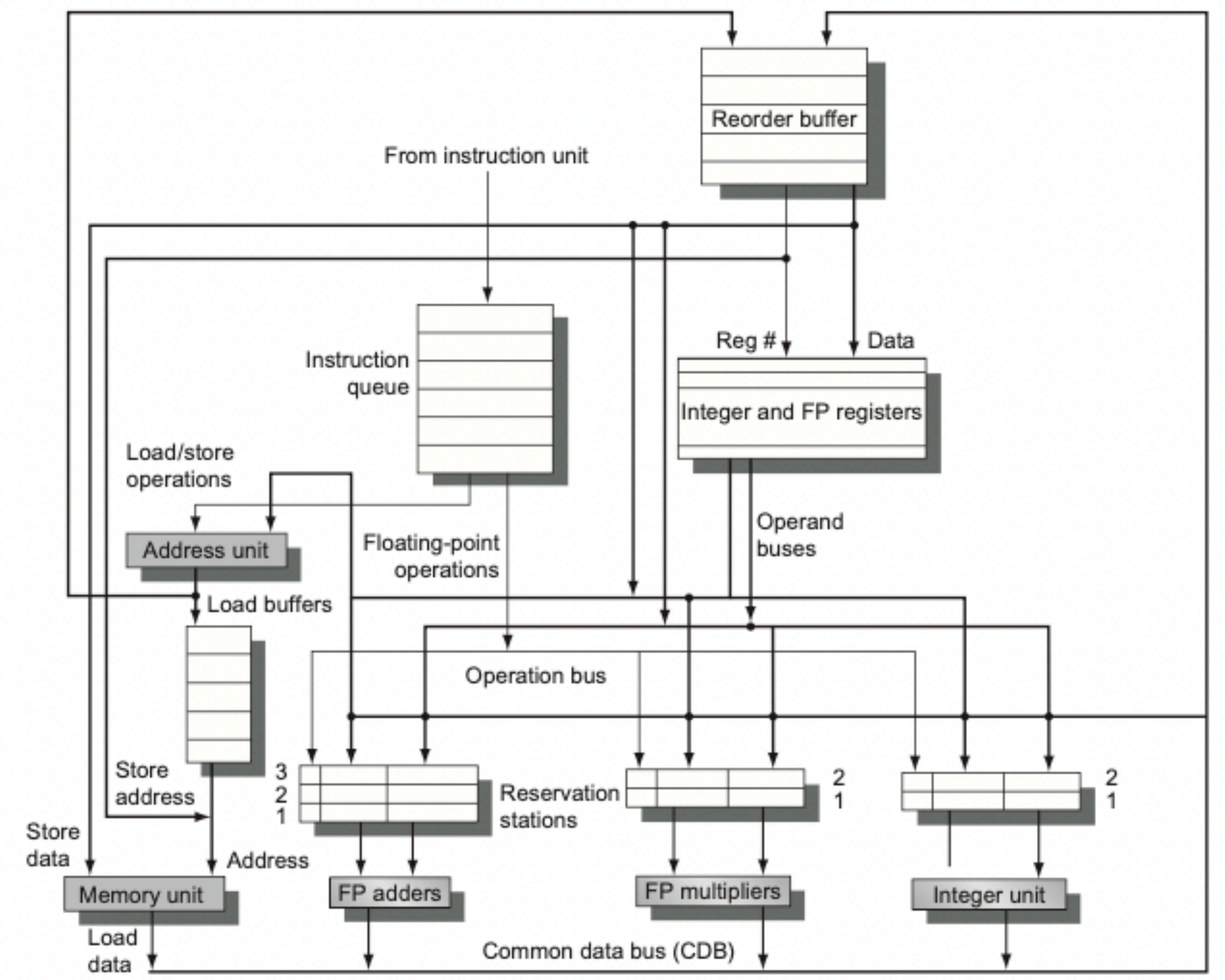

Figure 3.21 The basic organization of a multiple issue processor with speculation. In this case, the organization

Branch Target Buffer

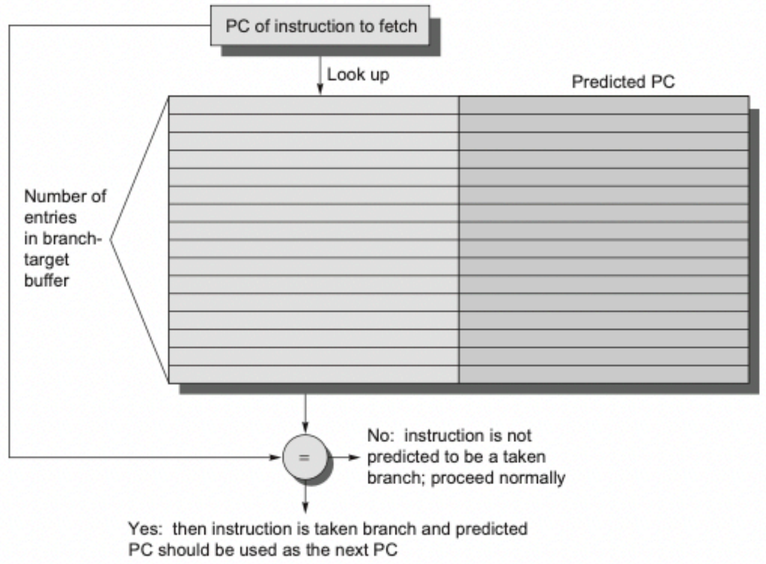

Figure 3.25 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one

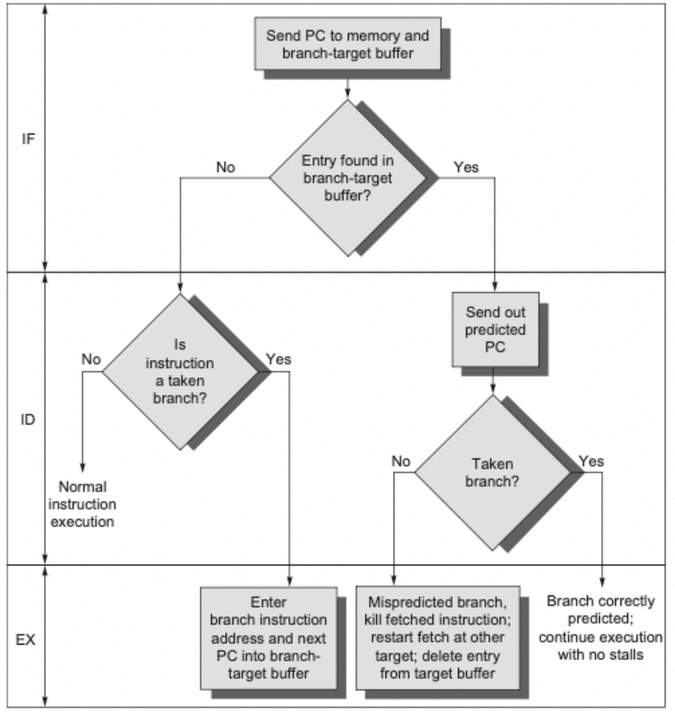

Figure 3.26 The steps involved in handling an instruction with a branch-target buffer.

Simultaneous Multithreading (SMT)

Intel calls this technique *Hyperthreading*

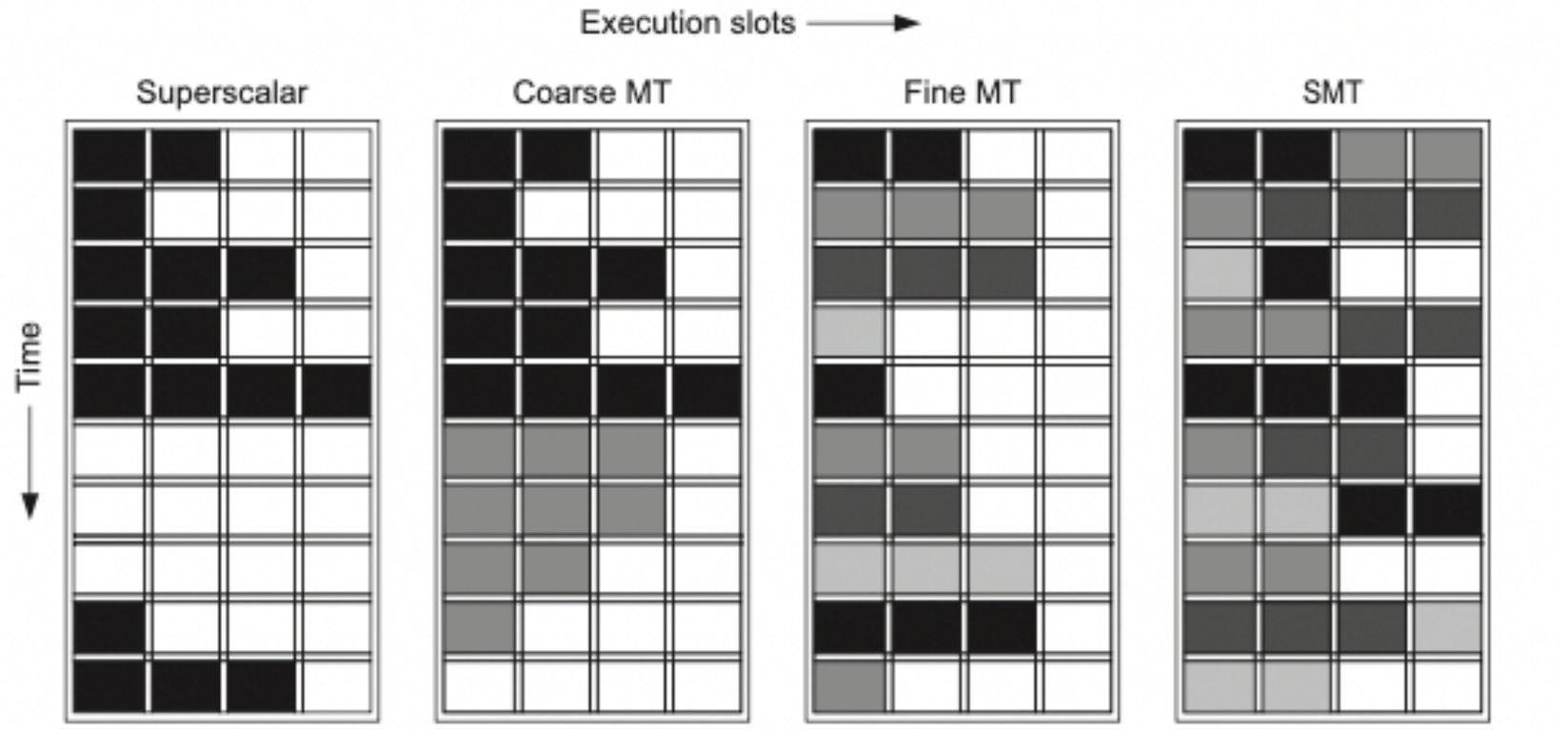

Figure 3.31 How four different approaches use the functional unit execution slots of a superscalar processor.

SMT Tomasulo Processor

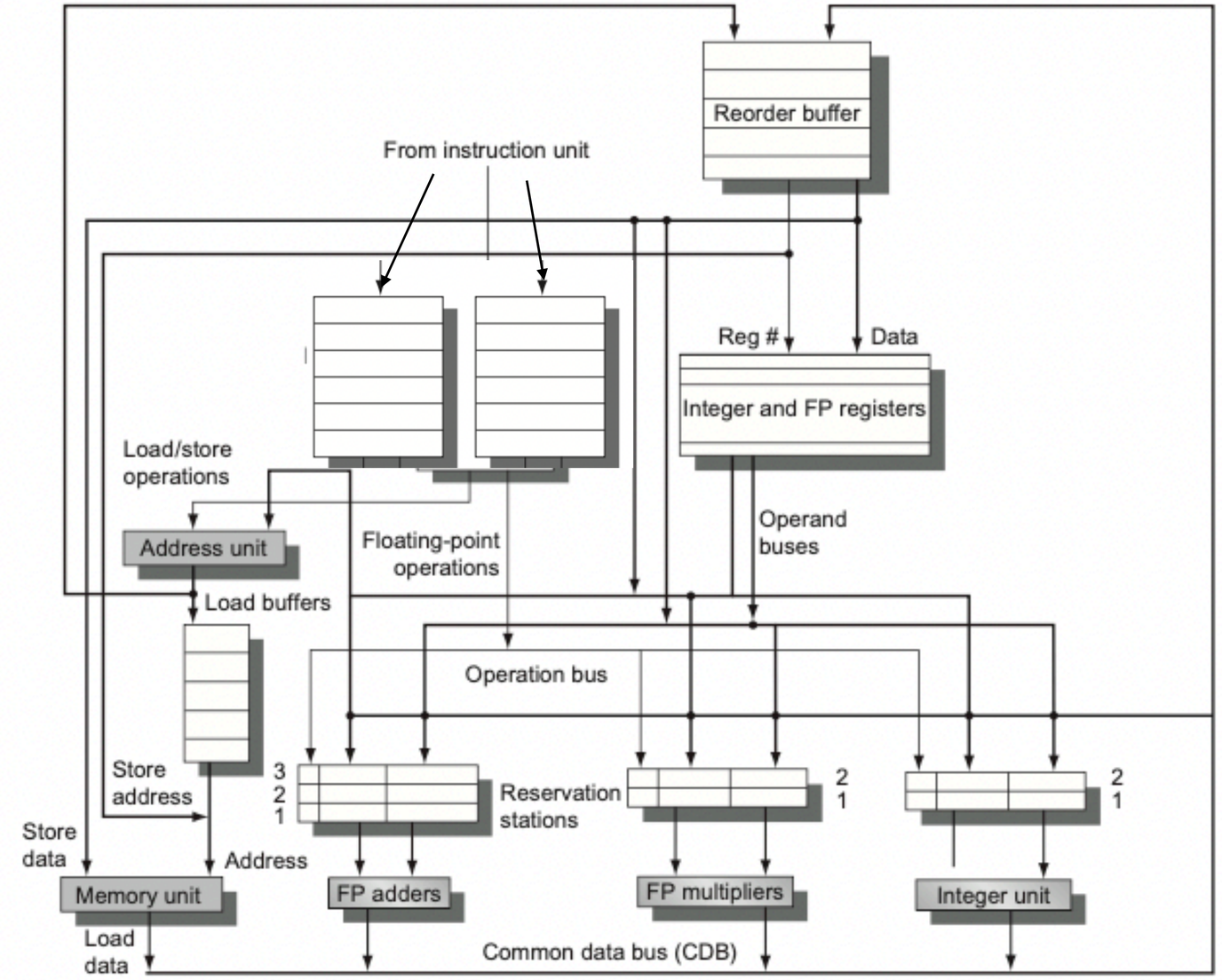

Figure 3.21 The basic organization of a multiple issue processor with speculation. In this case, the organization